Transforming Logic Programs By Specialising Interpreters

John Gallagher
SCS Technische Automation und Systeme GmbH
Oehleckerring 40
D-2000 Hamburg 62
West Germany

Abstract

A method of transforming logic programs by specialising meta interpreters is described. The transformation process requires three inputs; a logic program (the object program), an interpreter for some control strategy, expressed as a logic meta-program, and a query on the object program. The transformed program simulates the action of the meta-interpreter for the given object program and query, but much more efficiently than running the meta interpreter directly. The transformation may be viewed as compiling the control strategy into the object program. The transformation method is based on program specialisation; the transformed program is a specialised version of the interpreter which interprets only the given object program and query. It is suggested that this method of transformation can be useful in a problem solving system, by compiling general strategies into problem specifications.

Program Specialisation

To specialise a program means to restrict the input output relation which it computes. The aim of specialising a program is to obtain a version which treats fewer cases, but which deals more efficiently with those cases than would the general program. It is more efficient because some parts of the general computation can be precomputed in the specialised version, and because redundant parts of the general program may be omitted.

For logic programs, program specialisation may be characterised as follows. A logic program P defines a predicate r(x1,...,xn), which denotes a relation R consisting of the set of tuples (a1,..,an) such that r(a1,..,an) can be derived from the program. In a specialised version of P, the predicate r(x1,...,xn) denotes a subset of R. The subset of R may be specified by giving an instance of r(x1,...,xn) (that is, by giving some of the arguments of the predicate), or by adding constraints to r(x1,...xn)

r(x1,...,xn) & c(x1,...xn).

Methods of specialising programs have been developed by several researchers (Chang & Lee), (Goad), (Venken), (Bloch), (Komorowski), (Takeuchi & Furukawa), (Kahn), (Jones et al). All these use some form of partial evaluation or

symbolic evaluation of the program. As much as possible of the program is precomputed during symbolic evaluation, and the remainder, or residual program forms the specialised program.

Logic programming is particularly suitable for partial evaluation because of the generality of the unification mechanism. For pure logic programs, symbolic evaluation is no different from normal evaluation. However, partial evaluation of logic programs containing arithmetic or meta predicates need special symbolic evaluation mechanisms, which will be discussed below.

Although partial evaluation or symbolic evaluation are important, effective specialisation needs other techniques as well as these. In particular, it is sometimes necessary to construct new predicates in the specialised program; partial or symbolic evaluation alone cannot do this.

An Example of Specialisation

A logic program to transpose a matrix is given below. The matrix is represented by a list of its rows, where each row is a list of elements. The query

```
transpose(((1,2,3), (4,5,6)),Ans)
```

is solved with the substitution

```
Ans = ((1,4),(2,5),(3,6))
```

The program is as follows. The clauses are given references which will be used later. (X . Y) represents a list with head X and tail Y.

```
T1: transpose(M, ()) :- nullrows(M).

T2: transpose(M1, (Row.M2)) :-
        makerow(M1,Row, M3),
        transpose(M3,M2).

M1: makerow((),(),()).

M2: makerow(((X.R1) . M1), (X.Row), (R1.M2) ) :-
        makerow(M1, Row, M2).

N1: nullrows(()).

N2: nullrows((().M)) :-
        nullrows(M).
```

Program 1

Suppose the program is to be specialised for the case where the first row contains two elements, but the number of rows is unknown (greater than zero). The program is specialised with the query,

```
Q1:  transpose(((A,B).M1),M2)
```

The specialised program produced contains the clause,

```
transpose(((X,Y).M1),((X.R1),(Y.R2))) :-
     makerow(M1,R1,M2),
     makerow(M2,R2,M3),
     nullrows(M3).
```

Program 2

with the same clauses as above for makerow and nullrows.

If the number of rows is also known to be two, the query is

```
Q2:  transpose(((X,Y),R2),M2)
```

This gives rise to a completely determinate computation and the specialised program is the single clause,

```
transpose(((X,Y),(Z,W)),((X,Z),(Y,W))).
```

Program 3

The method of deriving these specialised programs will be described below.

Specialising Meta-programs

The transformation method of this paper depends on specialising meta-programs, such as interpreters. In principle, specialising a meta-program is no different from specialising an object program: the data just happen to be object programs. However, due to the lack of an explicit distinction between object and meta languages in most versions of Prolog, some particular problems need to be considered, such as the treatment of "meta logical" predicates supplied in Prolog. These will be discussed later.

The analogy between specialising metaprograms and compiling was identified by Futamura (Futamura). An interpreter may be abstractly considered as a function of two arguments:

```
Interpreter : (Program X Input) → Output
```

Compiling may be considered as a partial evaluation of the interpreter with the given program but no input (in particular, the partial evaluation parses the program) leaving a residual

program which is the compiled version, a function of one argument.

Compiled-program : Input → Output

The compiled program can be considered as a specialised version of the interpreter which interprets just the one program.

The significance of this analogy for logic programming is that there are many possible interpreters for logic programs, corresponding to different control strategies. It is prohibitively inefficient to run many of these interpreters directly, and impractical to write a separate compiler for each one. A general-purpose specialiser for logic (meta-) programs could compile any strategy for which a logic meta-program could be written.

The Specialisation Method

The query is symbolically executed, and a directed graph is constructed which represents a trace of the computation.

- The nodes of the graph represent states of the computation, that is, conjunctions of goals to be solved.
- The arcs are labelled by references to clauses of the program. A link from node A to node B labelled by clause C means that C is applied to the first goal in node A, giving the goals at B.

Using the clause references given in the above program, the graph produced for the first query Q1 is,

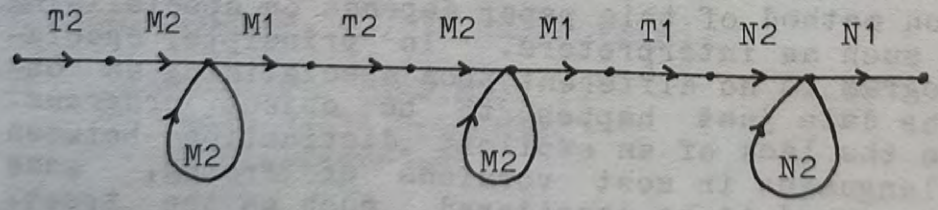

Figure 1

where the query is associated with the leftmost node.

The graph for the second query Q2 is,

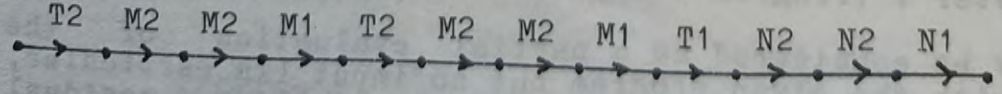

Figure 2

Construction of the Graph

An outline of how the graph may be produced now given. A description of a program to contruct such graphs may be found in (Gallagher). The graph is constructed by evaluating the query, adding an arc for each clause matched. One arc leaves a node for each clause which matches the first goal at that node. A crucial part of the process is obviously the detection of infinite paths in the graph; the query being evaluated may not have the required input which drives the computation towards termination. Various loop criteria are possible, but in each case it is necessary to identify some node in the graph with one of its predecessors (e.g. one test is whether a goal is identical with an ancestor except for renaming variables; this test, however, will not catch all loops).

- A branch node is a node with at least two arcs leaving it, which represent alternative course of computation.

- A loop node is a node with at least two arcs entering it, from which there is a path which leads back to it. (If a loop contains no branch node, the program would loop for all queries; the branch represents an exit from the loop.)

From the graph, the specialised program is constructed in two stages.

1. Extract from the graph all the clause sequences, which are the longest possible paths in the graph which do not pass through a loop node. (That is, any loop node in a clause sequence must be the last node).

2. For each clause sequence, construct one or more clauses for the specialised program.

The clause sequences are obtained by splitting the graph into sections, by detaching from each loop node all the arcs which lead into it. For the graph in Figure 1 this results in the sections,

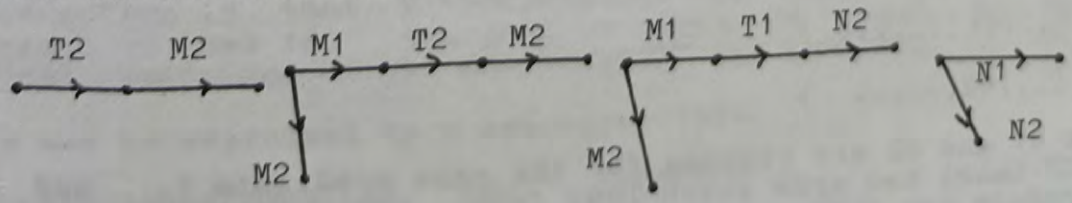

Figure 3

The graph in Figure 2 has no loop nodes, so cannot be split.

For each disjoint section of the graph, clause sequences are obtained by starting at a node with no arcs entering it, and tracing from it all paths ending with a node with no arc leaving it. From the sections in Figure 2 are obtained the sequences

```
(T2,M2)
(M2)  (M1,T2,M2)
(M2)  (M1,T1,N2)
(N2)  (N1)
```

Figure 4

A clause sequence gives rise to one or more clauses by applying the sequence to the most general goal which matches the first clause in the sequence. If the sequence is

$(C1,....,Cn)$ $n > 0$

then each Cj is applied to the head of the conjunction remaining after applying C1,...,Cj-1. It may happen that after applying some Cj in the sequence there are no more goals in the conjunction. In this case a new clause sequence is started from Cj+1.

The clause resulting from a sequence (C1,...,Cn), if G is the most general goal matching C1, is

$G' :- D1,..,Dk$ $k \geq 0$

where D1,..,Dk are the goals remaining after applying Cn, and G' is the instance of G which has arisen from the substitutions made in the process of applying the clauses.

The clauses obtained are given an ordering induced naturally by the clauses in the original program.

Renaming predicates

It is sometimes necessary to rename predicates in the specialised program, to distinguish calls to the same procedure from different places in the computation. If two clause sequences appear in the graph,

```
(...C1,..... )
(...C2,..... )
```

where C1 and C2 are clauses for the same predicate P, and C1 and C2 label two arcs which lead from two distinct nodes, we produce two different predicates corresponding to the different cases. In view of this renaming, the following clauses are produced from the clause sequences in the graph in Figure 3.

```
transpose(((X1.X2).X3),((X1.X4).X5)) :-
     makerow1(X3,X4,X6),
     transpose1((X2.X6),X5).

transpose1(((X1.X2).X3),((X1.X4).X5)) :-
     makerow2(X3,X4,X6),
     transpose2((X2.X6),X5).

transpose2((().X1), ()) :-
     nullrows1(X1).

makerow1((),(),()).

makerow1(((X1.X2).X3), (X1.X4), (X2.X5) ) :-
     makerow1(X3,X4,X5).

makerow2((),(),()).

makerow2(((X1.X2).X3), (X1.X4), (X2.X5) ) :-
     makerow2(X3,X4,X5).

nullrows(()).

nullrows(((().X1)) :-
     nullrows(X1).
```

Program 4

The program is refined by unfolding immediately all calls to procedures with only one clause, that is, to transpose1 and transpose2. Also, we note that makerow1 and makerow2 are identical, so they need not be distinguished. When these operations are done, we obtain the program given above (Program 2).

Interpreters for Logic Programs

Many authors have shown that interpreters for logic programs can be written concisely as logic programs themselves, e.g. (Pereira & Monteiro), (Porto), (Bowen & Kowalski), (Shapiro 83). The view of logic programming as deduction is that a computation is the proof of a theorem; that is, the goal or query is shown to be a logical consequnce of the sentences of the program.

This may be expressed by a meta-predicate,

```
provable("program", "goal", Subst)
```

where "program" and "goal" are the names of the program and the goal. That is, there is a naming scheme which represents the program and the goal as variable-free terms of first order logic. Subst is a substitution returned

if the goal is provable.

A logic program can be written (called INT), using Horn clauses with the standard procedural interpretation, which defines the predicate "provable" (Bowen & Kowalski). INT is an interpreter for logic programs; there are many versions of INT corresponding to different control strategies for the proof of the goal.

In fact, interpreters written in Prolog are often somewhat different from this.

1. The program is not represented explicitly as an argument of "provable", but is added as extra clauses to INT, in a form such as,

   clause(Head, Body)

2. Secondly, there is no explicit naming of object level terms as variable-free meta-terms, as above. There is no distinction between object and meta-level variables or terms. This means that the unification and renaming procedures built into the Prolog system can be used for the object level terms; otherwise as predicate such as

   unify(T1, T2, Subst)

   would have to be defined as part of INT, to unify object level expressions.

Some anomalies may arise because of the failure to distinguish object and meta-level, particularly with the so-called "meta--predicates" like var(X), X==Y, and so on. These predicates must be dealt with in the partial evaluation procedure, since meta-interpreters tend to make use of these predicates. However, a predicate such as var(X) is not really a logical predicate in Prolog; since for example var(X) may be true, but its instances false.

An Example of a Logic Program Interpreter

We consider the components of a coroutining interpreter. Coroutining in this sense means that goals in the same con junction which share variables are solved cooperatively. There are producers and consumers; a producer is evaluated until one or more specified variables are instantiated by a non-variable, after which it is suspended. A consumer is evaluated until one or more specified variables are about to be instantiated, after which it is suspended.

First the evaluator for the producer is defined. The predicate

```
produce(P1, P2, V1, V2)
```

means that the conjunction of goals P1 is solved (depth-first) until one of the variables in the list of variables V1 becomes instantiated by a non-variable. Then the computation is suspended, leaving a conjunction P2 waiting to be solved, and a new list of variables V2.

```
produce((),(), V,V).

produce((G.P), NewP, V, NewV) :-
     clause(G, Body)
          (newsubstitution(V, NewV),
           append(Body, P, NewP) ;

          no_newsubst(V),
          append(Body, P, P1),
          produce(P1, NewP, V, NewV)).
```

Program 5

where newsubstitution(V, NewV) is true if V contains non-variables and NewV is a list of the variables in V; no_newsubst(V) is true if V is a list of variables.

The clauses for consume are similar; the predicate

```
consume(P1, P2, V)
```

means that P1 is a conjunction which is evaluated until one of the variables in a list V is about to be instantiated by a non-variable. The computation is then suspended leaving a conjunction P2 remaining to be solved.

```
consume((), (), _).

consume((G.P), NewP, V) :-
     copy((G,V),(G1,V1)),
     clause(G1, Body),
          (newsubstitution(V1,_),
           NewP = (G.P) ;

          no_newsubst(V1),
          G = G1,
          append(Body, P, P1),
          consume(P1, NewP, V)).
```

Program 6

Note that the consume procedure copies the goal before matching it, so that it can check whether a variable in V becomes instantiated without actually performing the substitution.

A simple coroutining interpreter might then have the clause

```
coroutine(Producer, Consumer, Vars) :-
        produce(Producer, P1, Vars, V1),
        consume(Consumer, C1, V1)
        coroutine(P1, C1, V1).
```

Program 7

which calls the producer and consumer alternately. Many other variations are possible, with several producers and consumers; in the example below, two producers and no consumers are used. Using components like the ones sketched above, complex interpreters can easily be built up. They sometimes carry a lot of overhead to run them. This is the problem at which specialisation is aimed. It compiles-in the complex strategy with the object program, yielding a program which simulates the complex strategy when run under the standard control.

Comparing the Leaves of two Trees

The predicate leaves(T,L) means that L is a difference list of the leaves of binary tree T.

```
leaves( leaf(X), (X.Q) - Q).

leaves(tree( L, R), Leaves-Q) :-
        leaves(L, Leaves-LR),
        leaves(R, LR-Q).
```

Program 8

Given two trees T1 and T2, we wish to see whether they have the same leaves. A query to establish this is,

```
leaves(T1, L-()), leaves(T2, L-())
```

which succeeds if the leaves are the same. If the two trees differ, it would be advantageous to find out before generating the whole list for T1, as the standard control does. A sensible strategy is for each tree to produce one leaf at a time, which is then checked by the other tree, which in turn produces one leaf, and so on. Similar strategies are discussed in (Pereira & Monteiro) and (Burstall & Darlington). An interpreter to implement this strategy could be,

```
coroutine((),(),_).


coroutine(P1, P2, V) :-
```

```
produce(P1, NewP1, V, V1),
produce(P2, NewP2, V1, NewV)
coroutine(NewP1, NewP2, NewV).
```

Program 9

The query to this program is,

```
coroutine((leaves(T1, L-())),(leaves(T2,L-())), (L)).
```

where T1 and T2 are the trees to be compared, and the variable controlling coroutining is L.

Specialising this program (without knowing T1 and T2), we obtain the following clauses for produce:

```
produce((),(),V,V).
produce((leaves(leaf(X),(X.Q)-Q). G1),G2, (L), V1) :-
    var(L),
    produce(G1,G2,(L),V1).

produce((leaves(leaf(X),(X.Q)-Q).G1),G1,(L),(Q)) :-
    nonvar(L).

produce((leaves(tree(TL,TR),L-Q).G1),G2,(L1),V) :-
    produce((leaves(TL,L-LR),leaves(TL,LR-Q).G1),
                G2,(L1),V).
```

Program 10

The second clause corresponds to the absorption of a leaf which has been produced by the other producer, while the third corresponds to the production of a new leaf, after which the producer is suspended.

Partial Evaluation of Meta-Predicates

As mentioned above, meta-predicates cannot be treated like any other predicate. However, in order to derive the specialised program listed above, it is necessary to be able to evaluate these predicates. The solution we suggest is to distinguish different kinds of variables during partial evaluation. For many purposes, the following two types of variables are sufficient:

- A set C, containing variables which range over object level constant terms (i.e. containing no object level variables). These usually correspond to the "input" variables.
- A set V, which range over object level variables; these correspond to "output" variables.

The sets C and V are disjoint. In the example above, the

set C would initially consist of (T1,T2), the trees to be compared, and V would be (L), the output list of leaves.

After each unification, the two sets C and V may be altered. The rules for altering the sets, and for computing the values of meta-predicates, using these two sets, may be found in (Gallagher).

These rules allow the predicates newsubstitution(V,V1) and no_newsubst(V) in the produce and consume procedures to be evaluated symbolically.

The sets V and C may also be maintained by symbolic evaluation of arithmetic predicates like "X is Y", even when Y is not fully instantiated. A predicate,

```
evaluate((X is Y),V,C,V1,C1) :-
     constant(Y),
     append((X),C,C1),
     delete(X, V, V1).
```

expresses the fact that after evaluating (X is Y) where Y constains no variables, X becomes a constant and can be added to C and deleted from V.

Applications of Program Specialisation

Several authors have discussed useful applications of partial evaluation. It has been used for optimising database calls in a Prolog-database interface (Venken). A Prolog compiler has been produced by partially evaluating a Lisp interpreter for Prolog (Kahn). It may be used for type-checking (Bloch), and for compiling runtime checks for deadlock in Concurrent Prolog program (Shapiro 85), and for efficient control of expert system rules (Takeuchi & Furukawa).

The specialising of interpreters seems likely to give the most useful results. In particular, it can be exploited, as in the leaf-comparison example above, to transform a program running under the standard control strategy to one running under a user-defined strategy which may be quite complex and too expensive to use interpretively.

As another example, we have transformed an inefficient specification of the N-queens problem into an efficient version which places the queens one at a time and checks the safety of each one as it is placed (Gallagher). This was done by specialising a coroutining interpreter with breadth first search in the consumer. This problem has also been tackled recently by Bruynooghe et al. in a similar manner (Bruynooghe et al.).

Future Applications

In problem-solving, one is often required to solve a combinatorial problem whose solution requires the cooperative solution of a number of goals. Such problems are often easy to state as generate-and-test problems, but require complex strategies to solve them. It is suggested that specialisation of strategies can be a useful problem-solving technique. A problem-solving strategy sometimes uses much look-ahead to gather information about the search-space. This may often be done before run-time by specialising a strategy with a general statement of the problem to be solved. Thus general-purpose strategic knowledge can be intertwined with problem-specific knowledge.

Conclusion

A method of specialising logic programs has been described, based on partially evaluating the program, producing a graph from which the specialised program is derived. Unlike most other authors who deal with specialisation, we emphasise that partial evaluation is only part of the specialisation method; in particular it is sometimes necessary to create new predicates which were not in the original program.

The application of specialisation to meta-programs, especially interpreters, was discussed, and the analogy with compiling emphasised. Specialisation is presented as a practical method of compiling complex control strategies defined as meta-programs.

It is believed that specialisation of interpreters will be a useful method in logic problem-solving systems, by compiling a general strategy into a particular problem description.

Acknowledgements

Most of this work was done as a Ph.D. thesis in Trinity College, Dublin. I should like to thank Prof. J.G. Byrne, Hugh Gibbons and Mike Brady for useful discussions.

References

(Bloch)
C. Bloch, "Source-to-Source Transformations of Logic Programs", Weizmann Inst. of Science, Rehovot, Israel, Report CS84-22 (1984)

(Bowen & Kowalski)
Bowen, K. & Kowalski, R., "Amalgamating Language and metalanguage in Logic programming", in K. Clark and S. Tarnlund (eds.) Logic Programming, Academic Press, (1983)

(Bruynooghe et al.)

Bruynooghe, M., De Schreye, D., Krekels, B. "Compiling Control", Draft Report, Dept. of Computer Science, Katholieke Universitiet Leuven, Belgium, (1985)
(Chang & Lee)
Chang, C-L, & Lee, R.C-T., " Symbolic Logic and mechanical Theorem Proving", Academic Press (1973)
(Futamura)
Futamura, Y., "Partial Evaluation of Computer Programs, An approach to a Compiler-Compiler", J. Inst. Electronics and Communication Engineers (1971)
(Gallagher)
Gallagher, J., "An Approach To The Control of Logic Programs", Ph.D. thesis, Dept. of Computer Science, Trinity College, Dublin, Ireland (1983)
(Goad)
Goad, C., "Automatic Construction of Special Purpose programs", Proc. 6th Conf. on Automated Deduction, New York (1983), Springer Lecture Notes in Computer Science, Vol. 138
(Jones et al.)
Jones, N. et al., "An experiment in partial evaluation", Report 85-1, DIKU, Copenhagen Univ. (1985)
(Kahn)
Kahn, K. "A Partial Evaluator of Lisp Programs written in Prolog", Proc. 1st Int. Logic Progr. Conf., (1982)
(Komorowski)
Komorowski, H., "A specification of an abstract Prolog machine and its application to partial evaluation", Ph.D. thesis, Linkoping, (1981)
(Pereira and Monteiro)
Pereira, L.M., and Monteiro, L., "The semantics of parallelism and coroutining in logic programming", Colloquium on mathematical logic in computing science, Salgotarjan, Hungary (1979)
(Porto)
Porto, A., "Epilog: A language for extended programming in logic", Tech. Report, Dept. de Informatico, Univ. Nova de Lisboa, Portugal (1982)
(Shapiro 83)
Shapiro, E.Y., "A Subset of Concurrent Prolog and its Interpreter", Tech. Report, TR-003, ICOT, Tokyo, (1983)
(Shapiro 85)
Shapiro, E.Y., Lecture notes, Advanced Course in AI, Vignieu, France, (1985)
(Takeuchi & Furukawa)
Takeuchi, A., and Furukawa, K., "Partial Evaluation of Prolog programs and its application to Meta programming", ICOT Research Centre, Tokyo, Japan (1985)
(Venken)
Venken, R., "A Prolog meta-interpreter for partial evaluation and its application to source-to-source transformation and query optimisation", Proc. ECAI 84, Pisa, (1984)